



LIBRARY OF THE  
UNIVERSITY OF ILLINOIS  
AT URBANA-CHAMPAIGN

510.84

IL6r

no.469.474

cop.2







ILLIAC III REFERENCE MANUAL

VOLUME IV: Supervisor Organization

edited by

B. H. McCormick and B. J. Nordmann, Jr.

D. E. Atkins, R. T. Borovec, L. N. Goyal, L. M. Katoh,  
R. M. Lansford, J. C. Schwebel and V. G. Tareski

August 12, 1971



DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS



Report No. 472

ILLIAC III REFERENCE MANUAL

VOLUME IV: Supervisor Organization

edited by


B. H. McCormick and B. J. Nordmann, Jr.

D. E. Atkins, R. T. Borovec, L. N. Goyal, L. M. Katoh,  
R. M. Lansford, J. C. Schwebel and V. G. Tareski

August 12, 1971

Department of Computer Science  
University of Illinois  
Urbana, Illinois 61801

This work was supported by Contract AT(11-1)-1018 with the U. S. Atomic Energy Commission through September 30, 1970. Current support is under Contract AT(11-1)-2118 with the above agency.



Digitized by the Internet Archive  
in 2013

<http://archive.org/details/illiacciireferen473mcco>



## ABSTRACT

The Illiac III Reference Manual is being issued in this final documentation as four volumes:

Volume I: The Computer System

Volume II: Instruction Repertoire

Volume III: Input/Output

This issue-----Volume IV: Supervisor Organization

For ease of cross-reference an integrated table of contents will be issued separately.

The Supervisor of the Illiac III Operating System is principally concerned with task flow. Thus the Supervisor is directly responsible for time allocation, I/O requests, and other ongoing problems concerning individual tasks. This section (Volume IV) deals with the special hardware features of the Illiac III Operating System which facilitate the implementation of the Supervisor Program.



## ACKNOWLEDGMENTS

The Illiac III Reference Manual is based in part upon an earlier report:

- (1) B.H. McCormick (editor), William D. Bond, Kimio Ibuki, Roger E. Wiegel and John A. Wilber, Preliminary Programming Manual for the Illiac III Computer, Department of Computer Science Report 185, University of Illinois, July 1965.
- (2) B.H. McCormick and R.M. Lansford (editors), D.E. Atkins, R.T. Borovec, G.N. Cederquist, S.K. Chan, L.A. Dunn, J.P. Hayes, L.M. Katoch, P.L. Koo, B.J. Nordmann, Jr., J.A. Rohr, and J.C. Schwebel, Illiac III Programming Manual, Department of Computer Science Manual, University of Illinois, March 1968.

The present editors (B.H. McCormick and B.J. Nordmann, Jr.) acknowledge with gratitude the contribution of these earlier groups as transplanted to this greatly expanded manual. In addition, Dr. Rangaswamy Narasimhan contributed significantly to the early definition of the pattern articulation unit instruction set. In like manner Philip Merryman assisted materially in the early formulation of machine-implemented macros--the imprimitive instructions of Illiac III.

Authors and contributors to the manual are listed by principal area of concern:

Taxicrinic Processor:	B.J. Nordmann, Jr., R.M. Lansford, J.C. Schwebel, R.E. Wiegel
Input/Output Processor:	L.M. Katoch, V.G. Tareski, J.V. Wenta, G.M. Cederquist, J.P. Hayes
Arithmetic Units:	D.E. Atkins, L.N. Goyal
Pattern Articulation Unit:	R.T. Borovec, R.P. Harms, G.T. Lewis
Interrupt Unit:	L.M. Goyal
Exchange Net:	S.K. Chan, P. Krabbe
Scanner-Monitor-Video	L.A. Dunn, L.N. Goyal, V.G. Tareski, R.G. Martin, R.C. Amendola
Supervisor Organization:	B.J. Nordmann, Jr., R. M. Lansford

The seemingly endless drafts and revisions of this manual have been handled with patience and fortitude by Mrs. Betty Gunsalus and Mrs. Roberta Andre'. Illustrations were prepared by Mr. Stanley Zundo.



#### 4. OPERATING SYSTEM

##### 4.1 Operating System Structure

###### 4.1.1 Tasks and the Segment Tables

###### 4.1.2 The Supervisor Task

###### 4.1.3 Segments and Their Properties

##### 4.2 Inter-Segment Communication

###### 4.2.1 The Linkage Segment

###### 4.2.2 Supervisor Linkage Procedure

###### 4.2.3 Dynamic Segment Linking

##### 4.3 Inter-Task Communication

###### 4.3.1 Supervisor Calls

###### 4.3.2 Supervisor Manipulation of Task Segments

###### 4.3.3 Supervisor Return

##### 4.4 System Handling of Interrupts

###### 4.4.1 TP Operation Upon Interrupt Detection

###### 4.4.2 Interrupt Storage Segment

###### 4.4.3 Interrupt Handler Procedure

###### 4.4.4 Interrupt Return Operation



#### 4. OPERATING SYSTEM

The Illiac III Operating System is charged with the control of all program flow in the Illiac III system. It supervises all operations which the user tasks are either incapable of executing or not allowed to perform. In particular, the Operating System schedules tasks, allocates time and hardware and initiates all I/O operations.

The Illiac III Operating System has three major components: the Executive Program, the Supervisor Program and the Operating System Tables. The Executive Program is primarily concerned with job flow. Job flow involves the process whereby a given input job is broken down into tasks which in turn are scheduled for handling by the Supervisor Program. The Supervisor Program is principally concerned with task flow. Task flow involves the control of a given task during its operation. Thus the Supervisor is directly responsible for time allocation, I/O requests, and other ongoing problems concerning individual tasks. Eventually when the task is finished the Supervisor Program notifies the Executive which then proceeds to schedule the next job.

The Operating System Tables are the data base from which the Executive and Supervisor operate. Items in the Operating System Tables include the Segment Directory, the priority tables, the storage allocation tables, etc.

The TP itself is most directly concerned with the Supervisor Program since it is that portion of the Operating System which has direct contact with operating tasks. For this reason the remaining portion of this section will not deal further with the Executive Program.

The Taxicrinic Processors have been designed so that the operations of the Supervisor Program might be facilitated. With this in mind several instructions have been implemented which can be used by the supervisor to perform its assigned functions. Previously in Section 2 each supervisory instruction has been explained individually. The purpose of this section is to first describe, in minimum but adequate detail, the structure and operation of the Illiac III Supervisor Program. During this description the specific places where the supervisory instructions can be used will be identified. Any Operating System Tables whose description is necessary for a proper understanding of the Supervisor are also explained.

Several specific problems which arose in the Supervisor design are also discussed.



## 4.1 Operating System Structure

### 4.1.1 Tasks and the Segment Tables

The basic unit of programming in the Illiac III System is the task. A task consists of a set of program and data segments related to the performance of a specific programming job. Associated with each task is a Segment Table consisting of base descriptors for the segments used by the task and a Status Block. The actual entry position within the segment table for a given segment's base descriptor is taken as the internal segment name for that segment. This internal segment name is, of course, local to the task in question.

From a software standpoint, each segment table is divided into three regions: the Interrupt Entry, the Global Segment Table (GST), and the Local Segment Table (LST). The Interrupt Entry points to a circular storage buffer used to save the status of the machine during the various levels of interrupts (see Section 4.4.2). The Global Segment Table lists the base descriptors for those segments common to all tasks at a given level in the system, i.e. system programs, certain bookkeeping routines and other various common subroutines. The Local Segment Table lists base information for those segments specific to the given task. It should be noted that this distinction between the latter two areas in the segment table is purely a software convention.

As shown in Figure 4.1.1, the tasks themselves can be thought of as being arranged in a hierarchy. At any given node of the task hierarchy, all of the descendant tasks may possess identical global entries in their segment tables. These global entries are, in fact, identical to the combined local and global entries in the segment table of the parent task. This type of structure arises from the fact that

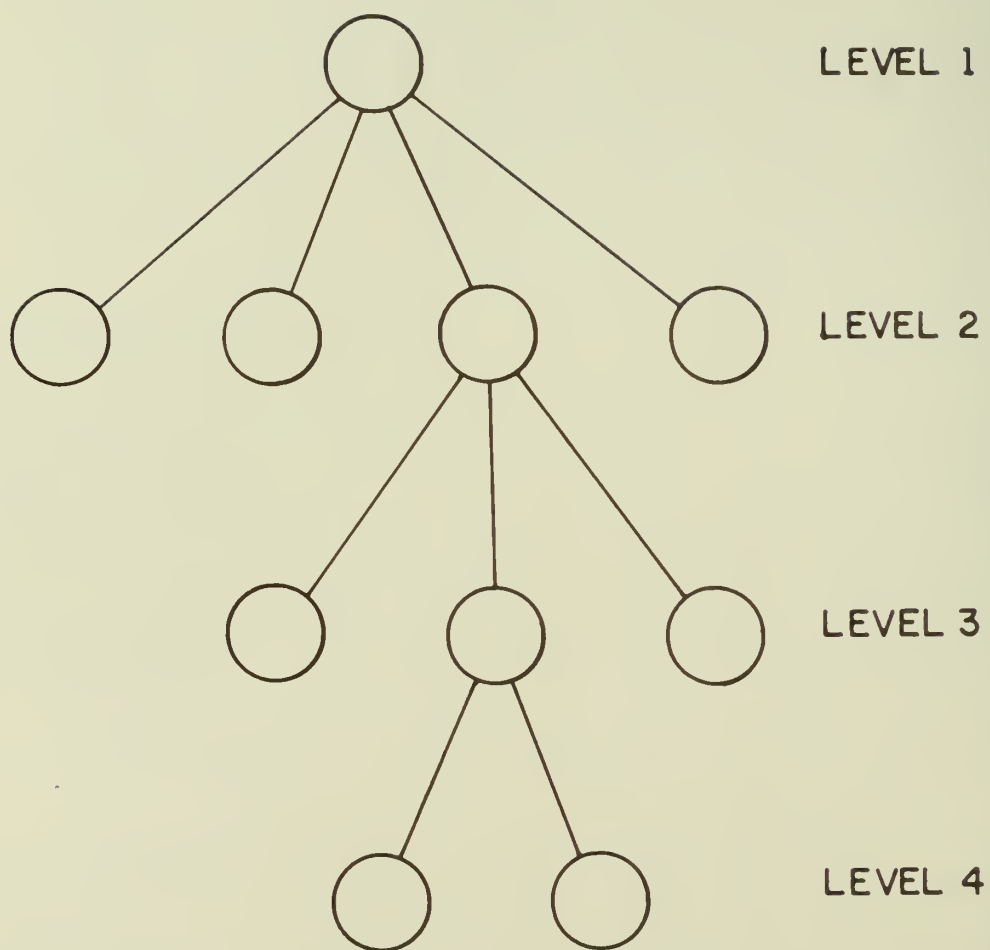


Figure 4.1.1 - Task Hierarchy

although the system may start with a fixed number of tasks, it is possible for these tasks, in turn, to initiate other subsidiary tasks. These derived tasks may, in general, contain not only segment table entries to all of the segments used by the parent task but also entries for segments used only by the sibling task.

Thus, all of the sibling tasks derived from a single parent task, share identical global segment areas. It is important to note that in this type of segment sharing each task calls the shared segments by the same internal name, i.e. the same relative entry location is used in the segment tables of all tasks sharing the segment. However, a mechanism must also be provided for tasks to use the same segment and yet not call it by the same internal name. This occurs when two different tasks discover a need for the same segment but at different relative times in the history of their task execution. In this case, different entries in their segment tables will be used. But since the contents of those two entries are identical the segments used will be the same, though they will be known by different internal segment names within the two tasks.

One interesting technique which can be used in creating tasks which share large numbers of Segment Table entries is to use Partitioned Mode accessing. Normally, a new descendant task is initiated by simply duplicating the parent's segment table and then adding at the end any segments specifically used by the descendant task. However, if space is at a premium, the new segment table can be stored in partitioned-mode. Then core need only be allocated for the page map, the page in the new task's segment table which holds the Interrupt Entry (i.e. the base descriptor in the segment table), and any other pages which are not held in common with the parent task. Hopefully this latter category will not be large and thus a considerable space savings might be accomplished.

As to the actual segment table entries themselves, each entry requires one double word. The first three bytes comprise the same information which is placed in the TP base registers, i.e. the bounds, the base address, the partitioned/contiguous mode flag, and the 2 access privilege mode flags. The fourth byte contains miscellaneous information for the Operating System. The second word in the segment table entry consists of information necessary to locate the main directory entry corresponding to the given segment. This is normally a pointer to a directory index, which in turn will contain a pointer to the actual directory entry.

## 4.1.2 The Supervisor Task

The Supervisor is run as the zeroth level task. Thus like any other task it has a segment table consisting of an Interrupt Entry, Global Segment Table and a Local Segment Table. Figure 4.1.2 shows the structure of the Supervisor task segment table.

Note that in addition to the above-mentioned standard three constituents, the Supervisor task segment table also has a fourth area called the Dynamic Segment List. This region is used to hold base descriptors for segments which the Supervisor has particular interest in at any given moment. It contains base descriptors for the segment table of every task known to the Supervisor (task name entries). Also included are ordinary procedure or data segments in which it is especially interested (e.g. blocks containing I/O information for a particular task). Note that since the number of local segments in the Supervisor Segment Table will not change dynamically, the initial position of the Dynamic Segments List within the Supervisor Segment Table is a program constant for any particular version of the Supervisor.

The Dynamic Segment List is called "dynamic" because, unlike other segment table entries, the entries in this list may be discarded when the Supervisor no longer has need for them, and can be later reassigned to other segments. The control of these dynamic entries is effected through the use of an Available Space Pointer. Initially, all the entries in the Dynamic Segment List are placed on the free list of an Available Space PR or in its contiguous storage area. Then whenever a new entry is needed, the GET instruction can be used. Then when the entry is no longer needed the PUT instruction can be used

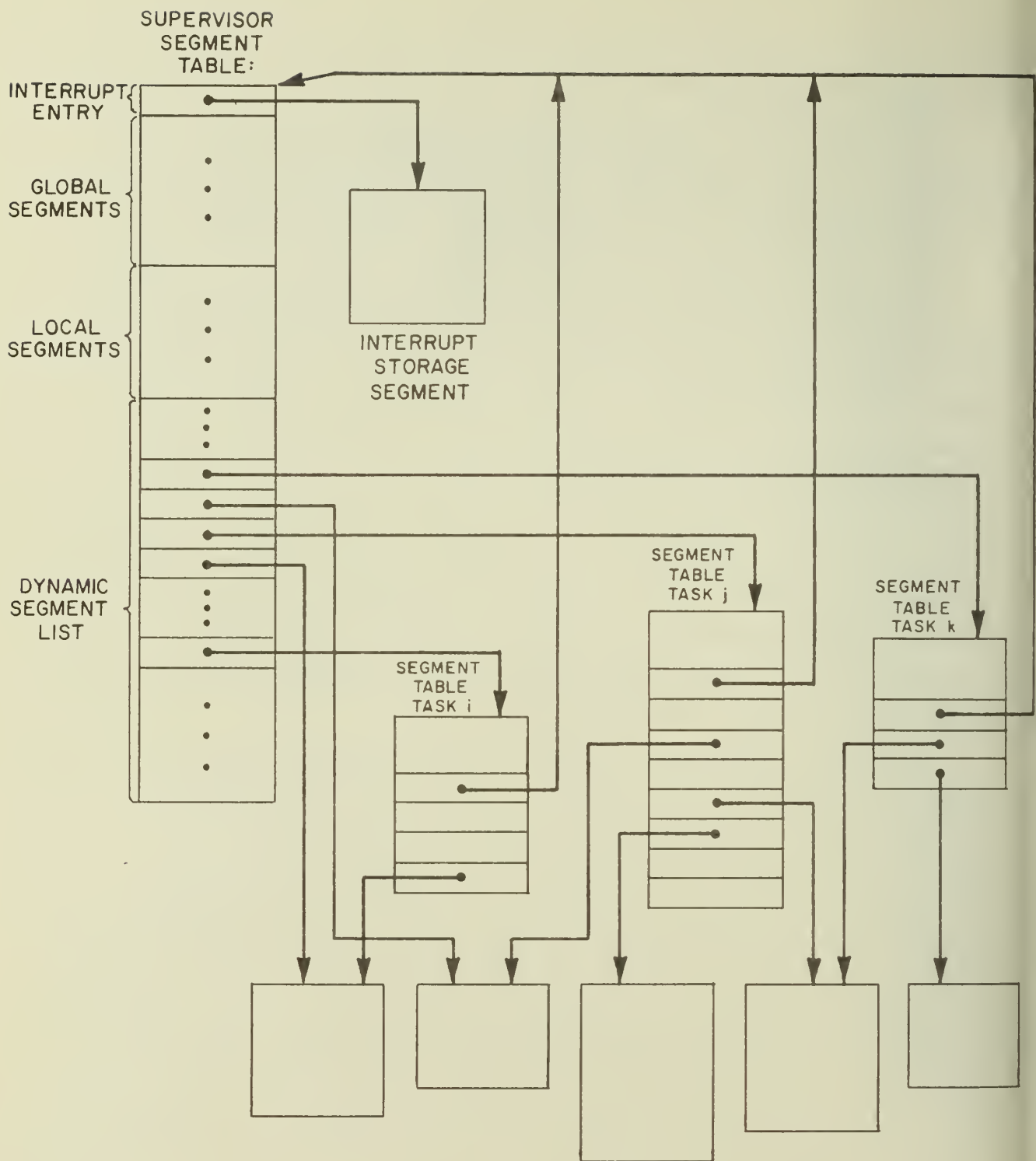


Figure 4.1.2 Supervisor Task Segment Table

8/10/71

Section 4.1.2 2/4



to return it to Available Space. Note that the entries are on a linked list when not in use; they are not on a list when they are actively being used to hold base descriptors.\*

As described above, entries in the Dynamic Segment List are created as needed. A new entry is made every time the Supervisor is given a new task to process by the Executive Program; here the base description of the task's segment table is entered. The address of the new entry within the supervisor segment table becomes the task name for the new task and remains constant for the duration of the life of the task. The method whereby other types of segment base descriptors may be added to the Dynamic Segment List is explained in detail in Section 4.3.2.

It should also be noted that although the ordinary segment entries in the Dynamic Segment List are in every way identical to segment entries in the rest of the Segment Table (see Section 4.1.1), the entries for segment tables themselves (i.e. task name entries) contain the segment name and internal address of the task's Status Block in the second word of the double word entries instead of a Main Segment Directory pointer. This is useful since it gives a simple way of finding the SB of any desired task. The Main Segment Directory pointer for each task's segment table is found in that task's SB.

---

\*It should be noted in passing that Available Space techniques will be useful at several places in the Supervisor. It is anticipated that each use will probably involve a different segment to be parcelled out as cells. Thus it rapidly becomes impractical to use a new PR for each new available space segment which is needed. What is probably more practical is to keep the contents of the various available space pointer registers in some data area in core and to load the required PR image (in the available space format) into a PR whenever it is needed.

In the case discussed above, it would be preferable to use PR#14 since when this register is used the hardware automatically supplies double word cells. However, if this is not possible, and it becomes necessary to use another PR, then it will be necessary to reserve the zeroth location in the supervisor task's segment table for the cell size of the dynamic entries. This will allow the available space hardware to automatically obtain the correct size cells. If it is not possible to reserve the zeroth location it will be necessary to initially partition the area assigned to the Dynamic Segment List into linked double word cells contained on a free list.

An important problem encountered in the design of the Supervisor concerns the fact that the Supervisor can actually run on more than one processor at a time. Situations arise where it may be necessary to prevent access to the same data by two (or more) processors either at the same time or within a few micro-seconds of one another. A typical example of this arises in updating a system table. During the time that one processor is updating the table, the table will not contain proper data and thus other processors should be prevented from accessing this data until the updating process has been completed.

The procedure for insuring this "locking out" of competing processors makes use of the INCK instruction. This instruction can be used to access, update, and replace a control word without permitting another processor to modify the control word in the interim. Thus by using a control word to indicate whether a processor is using a certain "critical section", and having all processors check this word before entering the section, the system can ensure that only one processor will be in the critical section at one time. The exact procedure used to "lock out" processors is explained in detail in Section 4.4.2.



### 4.1.3 Segments and Their Properties

A segment is a collection of information sufficiently important to warrant being assigned a name, i.e. a location in one of the Segment Tables. The segment may consist of procedure code or other data base information and can be retained in either contiguous or partitioned storage format in core. When not in core, the segment may be stored on a secondary storage device (disk, drum, etc.) in any format consistent with that type of storage.

When in core, a segment consists of an integral number ( $\leq 256$ ) of pages. During execution, the size of the segment may be changed by an integral number of pages. Although, as mentioned above, a segment may be stored in either contiguous or partitioned format, this distinction will be ignored for the remaining discussion about segments since the procedures necessary for handling partitioned mode are entirely hidden from the programmer.

There are several properties of segments in the Illiac III system which should be kept in mind in the discussion of supervisory functions:

- 1) Most procedure segments will be written in reentrant code. In particular, all system programs will be reentrant. This means that the procedure will not modify itself and thus one copy of the code can be used by several tasks concurrently. It also means that the external segments referenced by the reentrant segment will be translated into different segment table addresses for each task using the reentrant segment. Thus, the segment table names cannot appear within the procedure code itself and each task using the procedure must have a different linkage table for translating the external segment name into an entry address in the task's segment table.

2) Segments should only be loaded and linked when they are actually needed. This process is known as dynamic linking and is desirable since it avoids filling the computer memory with segments which might not be used by the task.

3) Segments must be relocatable to new areas of core. If one task is temporarily idle due to an interrupt or I/O wait, it may be necessary to remove some of its segments to provide more core space to currently executing tasks on the TP's. When the interrupted task returns, its segments will need to be reloaded -- in general into a different area of core. This must be done with a minimum of confusion for the segments already in core.

## 4.2 Inter-Segment Communication

Inter-segment communication refers to the process whereby one segment in a task either fetches from, stores into or transfers control to, another segment in the same task. The problem here is to determine whether or not the second segment is in core and if so, where it is located.

In conventional batch processing all segments of a job are specified at the beginning of the job. During the loading process each segment is told by means of a transfer vector or other such technique where each of the other segments it needs are located. This is known as the linking process.

In the Illiac III system a task may not necessarily have available, at the time it is activated, an explicit list of the segments it will be using. The supervisor may only learn that a certain segment is part of the given task at the time the task first reaches a point in its execution where it actually requires that the segment be in core. Prior to this moment the segment is simply an anonymous file somewhere in secondary storage. The purpose of this section is to describe how the process of dynamic linking takes place and to demonstrate some of the supervisory instructions used to implement it.

It is implicit in the following discussions that each segment utilizing external references contains an external name list. Consecutive entries in this list are the 8-character (doubleword) names of the external references.



#### 4.2.1 The Linkage Segment

Every task which runs on a Taxicrinic Processor must have a linkage segment. The task's linkage segment is an integral part of the process of intersegment communication since it provides the means for linking an external reference identified at compilation time to the corresponding pointer (internal segment name and displacement). These pointers are idiosyncratic to the task, i.e. two distinct tasks can point to the same external reference only by using in general different pointers (as the internal segment names refer here to two different task segment tables).

The linkage segment itself consists of an index and various subtables called linkage tables. There is one linkage table for each segment in the given task which utilizes external references.

The linkage segment index consists of a double word entry for each table within the linkage segment. The left halfword, or link field, contains the entry point within the linkage segment of the corresponding linkage table. The adjacent two halfword fields of the index entry contains a pointer (internal segment name and displacement) to the list of external names to be linked. Each external name (segment with/without entry) is described by an 8-character string value in this latter list.

The linkage tables within the linkage segment consist of one word entries. Each table has one entry for every external name referenced by its corresponding segment. The leftmost flag is the link/no-link indicator. If it is '0' the entry has not yet been loaded with a pointer to the external reference itself; rather, the table entry contains a pointer to the location of the external name (i.e., the corresponding 8-character cell within the External Name List.

If the link/no-link flag is '1', the entry has already been linked: i.e., a pointer to the external reference has been developed. In this case the right halfword contains the internal segment name and the left halfword contains the displacement of the entry point in the external segment represented by that particular external name. (If the external name represents a segment, then this left halfword will be zero.)

The linkage table is set up either when the segment is first entered into the segment table--if the segment is in core at the time the new entry is made, or when the segment is first read into core--if it had not previously been resident in core. In the latter case, an entry is initially made in the linkage segment index with a pointer filled with zeros. Whenever space is allocated for the linkage table the first section of the segment is scanned to determine how many external names are needed.<sup>1</sup> Then space is allocated in the linkage segment for a linkage table of the proper size. Also, the link portion of the proper index entry is filled with the beginning address of the new linkage table.

It is important to note two things at this point. First, since the linkage segment index must appear at the beginning of the linkage segment and since its length will not be known when it is first set up, it will be necessary to allocate enough storage for the index at the beginning of the task so that there will be room for the most probable number of entries. At the same time, economy demands that this amount of storage not be excessive. If the allotted storage is eventually filled and more room is needed, there will be little recourse except to repack the segment, which necessitates a tremendous amount of adjusting in the index pointers since all of the linkage tables must be shifted. Alternatively, partitioned mode can be used. Here linkage tables can be created beginning with the highest segment virtual addresses; pages would be assigned to the page map only as needed (i.e. the center pages of the segment would initially be nonexistent).

Once storage has been allocated for the linkage table and the index entry has been loaded, all of the entries in the table are initially loaded with the pointers to the segment making the external references. (Later in the linking process each of these pointers will be replaced by the pointer of the external reference itself.)

---

<sup>1</sup>The scan could be eliminated by having a count at the beginning of the segment.



The segment corresponding to a particular linkage table has the format shown in Figure 4.2.1/1. The first section consists of a list of double word entries containing the first 8 characters in the names of each external name referenced by the segment. This list corresponds entry-for-entry with the linkage table. Thus the  $j$ th doubleword entry in the external segment name list contains the 8-character external segment name of the segment whose internal segment name is in the  $j$ th word entry of the corresponding linkage table.

The next segment section is the prologue which consists of various code sequences which must be executed before the actual procedure can begin. Finally, comes the code for the procedure itself.

It should be noted that when filled, the entries in the linkage tables link the external references in the given segment to entries in the task's segment table. Thus, as will be seen later, the linkage process need only be performed once during the lifetime of the task. For once an entry has been made in the task's segment table, it remains at the same relative address for the life of the task. Even if the segment is later paged out and still later brought back in, the old linkage will still be valid: although the contents of the segment table entry for the newly moved segment will be different, its location (and thus its internal segment name) will still be the same.

Linkage Segment:

PR Value 1	Seg. No. 1
PR Value 2	Seg. No. 2
.	
.	
.	
PR Value N	Seg. No. N

PR Value and  
Seg. No. fields  
Consist of a  
Halfword each

Procedure Segment:

<Seg. Name 1>
<Seg. Name 2>
<Seg. Name N>
Prologue
Segment Code

External Segment  
Name List: each  
doubleword entry  
consists of 8  
characters

Figure 4.2.1/1 Dynamic Linkage Table Format



#### 4.2.2 Supervisor Linkage Procedure

The basic linking process is performed by the Linkage Procedure of the supervisor. The purpose of this section is to outline the operation of this procedure.

The goals of the linkage procedure are severalfold:

- 1) On the basis of an eight-character external name and task name, it must determine exactly which segment (and entry point within the segment) is desired.
- 2) It must ascertain whether the segment has already been assigned to the task and if so, what the internal segment name is for the requesting task.
- 3) If the segment has not previously been assigned, it must check the access privilege tables within the system to insure that the task is allowed access to the segment in question.
- 4) If the task is allowed use of the segment, an entry into the task's segment table must be provided and loaded with the proper access mode and other descriptor data. At the same time the Linkage Procedure must determine if the new segment utilizes external references and if so it must provide for a linkage table and an entry pointer to the table in the linkage segment index.
- 5) The system segment directory must be updated to indicate that an additional task is using the segment. In addition, various other system tables may need to be updated.

To accomplish these aims the linkage procedure must be given as parameters both an 8-character representation of the external name to be linked and the task name. By adding the task name, a unique name is generated.

The Linkage Procedure takes the name given to it as a parameter and searches the directory name table until it either determines that the name is not in the table or finds out where the main entry is.

If the external name exists in the directory name table, the linkage procedure next goes to the directory index and then to the directory entry itself to ascertain whether the corresponding segment has already been assigned to the requesting task. This can be implemented by scanning a list of pointers to the various segment tables using the segment. This list will undoubtedly also be needed in order to change segment table entries whenever a segment's status in core changes.

If the segment has already been assigned to the task, the linkage procedure can obtain its internal name for that particular task (the appropriate displacement for the entry in question), and return. Otherwise, the linkage procedure must further check the access restrictions on the requested segment to see if the given task is allowed to access the segment. If so, the Linkage Procedure loads the task's segment table with the new entry. Then it must check the directory to see if the segment utilizes external references and, if so, create an entry in the task's linkage segment index. Both the address and length of its external name list must be determined. (It may be most practical to simply read in the segment at this point if it is not already in core.) With this information the corresponding linkage table can also be filled in one-to-one with pointers to the entries in the segment's external name list. Finally, the linkage procedure makes all of the required updates in the other system tables.

It should be noted that if for some reason an improper name is given to the Linkage Procedure, one of three things might happen:

- 1) The name is not found in the directory name table. Here a message so stating will be returned to the user.
- 2) The name is found in the table but the user will not be allowed access because the segment associated with that name is prohibited from his use. In this case he will get an error message.

- 3) The name matches one which he is allowed to use.  
Here the Linkage Procedure will proceed normally.  
The system however has no way of knowing whether the user really wanted this name or even that a mistaken identification has been made--in either case the Linkage Procedure is set in motion.



### 4.2.3 Dynamic Segment Linking

The purpose of this section is to describe the general process involved in linking up a segment's external references to the actual segments they represent. Figure 4.2.3 shows the relationship between the various system tables and links.

When a given procedure segment is entered, the first code executed is the Prologue, which sets up various parameters and allocates storage for the procedure. In addition, the prologue has to set up a pointer to the linkage table corresponding to the given segment.

This latter is done by loading the standard name for the linkage segment into the PR which will be used to access the linkage table. The actual virtual address of the linkage table in question can be found by then searching the name fields in the linkage segment index. This search could be performed either with a binary search method if the index is long, or using the SCANM (Scan with Mask) instruction. However the search is performed, the final virtual address is stored in the pointer value field of the PR designated as the Linkage Table Pointer. This PR will then remain known to the procedure segment for the remaining time that the segment is in control of the process. Once the Linkage Table Pointer has been loaded, all references to segments external to the given procedure can be enacted using the LINK instruction.

All memory accesses in the Illiac III system must be made through Pointer Registers; accordingly to link an external segment it is only necessary to get the PR's loaded with the proper contents. It is necessary then during compilation and/or assembly to group all external references (segment names with/without entry points) into a single table within the segment. Each external name is referenced by a LINK instruction, sometime prior to the use of the name. The compiler (or assembler) is responsible for seeing that the operand in the LINK instruction referring to the external name contains a canonical address. This address is, in fact, the relative location of the one word cell in the linkage table which contains the internal segment name and entry point for the corresponding external reference: i.e., a pointer to the external referenced call.

Thus at every point in the segment where it is necessary to load a PR with an external reference whose segment name and/or

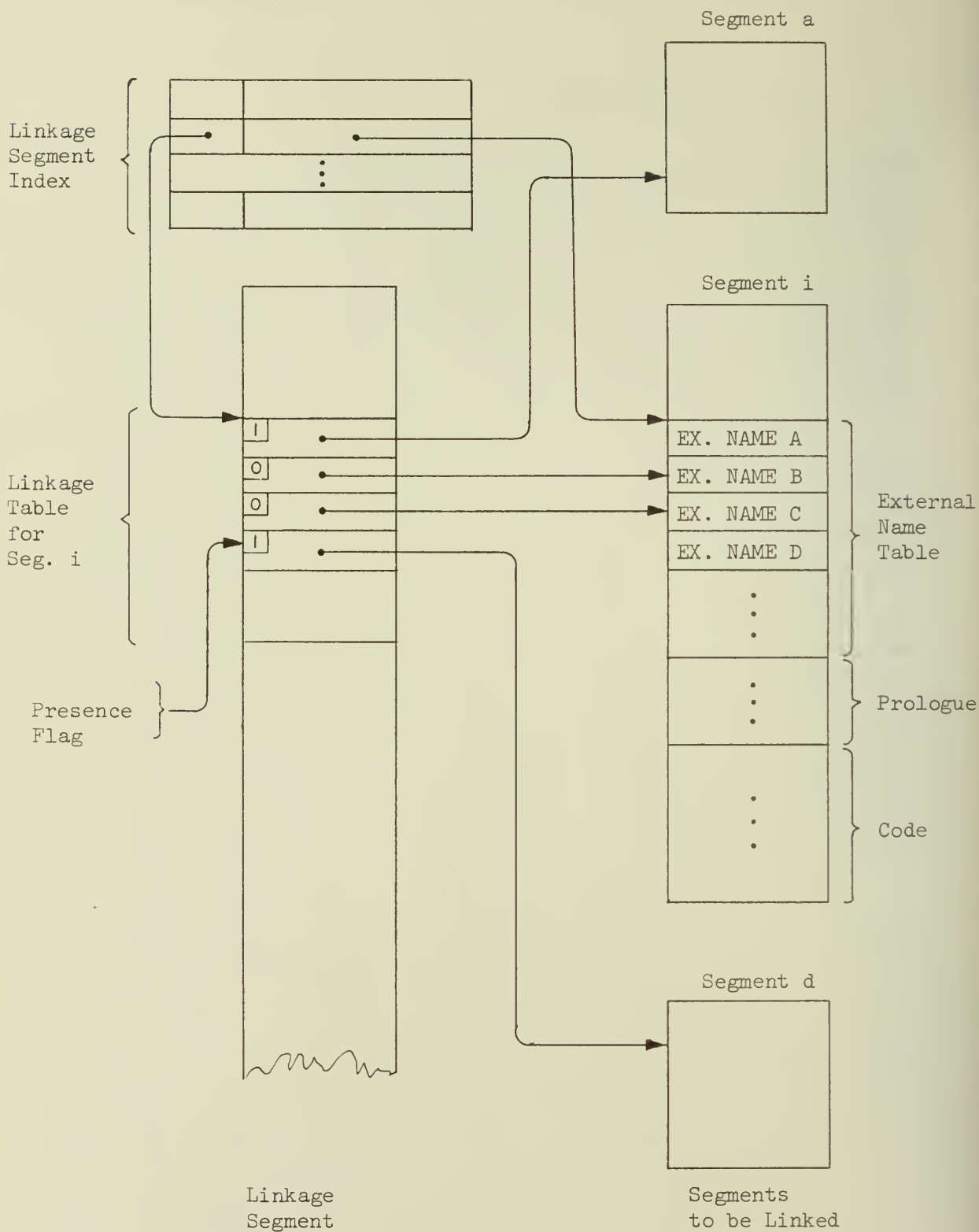


Figure 4.2.3 Segment Linking Tables



entry point is not presently known, the PR is first loaded with the respective canonical virtual address by means of a replacement pointer register pre-operation. Then the LINK instruction is executed using as operands (1) the Linkage Table Pointer (previously loaded by the Prologue code) and (2) the PR which is to be linked.

During the execution of LINK, if the segment entry in the linkage table has already been filled, the link/no-link flag will be '1'. In this case, the PR designated by the instruction is loaded with the segment name and pointer value found in the entry, and the procedure continues.

If the link/no-link flag is '0', then the entry in the linkage table for that particular external segment has not been filled. In this case, a linkage interrupt is performed and the Interrupt Handler, in turn, calls the linkage supervisor program.

Before calling the Supervisor, the Interrupt Handler has the responsibility for supplying the information necessary for the Supervisor Linkage Procedure to find the right segment (see Section .2.2). At the same time the Interrupt Handler is restricted to using only information which was supplied with the interrupt. One of these items is the internal segment name and virtual address of the address which caused the interrupt. Since in the case of a linkage interrupt this address is always the address of the particular entry in the linkage table which caused the interrupt, it is fairly straightforward to obtain the necessary information.

At the time a segment is being assigned a linkage table and an entry in the linkage index it is possible to fill each unlinked entry in the linkage table with pointers to the new segment's External Name List. In particular, if the External Name List is contiguous (in virtual memory) each unlinked entry in the linkage table can be made to point directly to its corresponding entry in the External Name List simply by starting with the initial address of the list and incrementing each entry by 8. If the address of a segment's External Name List and its length are stored as part of the information in the general Segment

Directory there would not even be a need to read the segment into core before performing this operation.

Thus when a linkage interrupt occurs, it is possible to obtain the 8-character external name which caused the interrupt simply by accessing the address which is stored in the location pointed to by the interrupt address. The task name is easily obtained by the Supervisor from the Task Register.

Eventually the Supervisor Linkage Procedure will return the segment name and virtual address of the external reference. Then the Interrupt Handler simply replaces the contents of the unlinked entry with these new pointer fields, sets the link/no link flag to '1' (i.e. link) and returns so that the LINK instruction can continue. Thus by using this instruction it is possible to load the internal segment name and virtual address of any desired external segment entry point into a PR.

Note that if a given link operation is the first to use a given segment, the segment will be assigned an entry in the segment table and be linked up from the requesting segment. However, the new segment itself may still be on secondary storage (unless some other task was using it and left it in core somewhere) and will not be called into core until a memory access to that segment is actually attempted. This can keep the TP from wasting time if it later turns out that the segment was not used.

One question which arises from this dynamic linking scheme is whether or not linking within the Supervisor Task itself can be handled with the same techniques. The answer is that in general it can although there is at least one important exception. Provided that all of the Supervisor Task segments are given names and entries in the Main Segment Directory Table, there is no basic problem when a given Supervisor segment needs to access some external segment for which it has an 8-character external name. The exception involves the Supervisor Linkage Procedure itself. If this procedure ever tried to link itself to an external segment and discovered that the segment was not there, it would



have to call itself and thus get into a never-ending loop. The solution is to make sure to load up the Linkage Table for the Supervisor Linkage Procedure at System Generation time. This should not involve too much work since the Supervisor Task's Segment Table will already contain all of the global and local segments it will ever use and, in addition, the external segments used by the Supervisor Linkage Procedure will probably have standardized locations within the Segment Table. In general, for efficiency's sake, it would probably be best anyway to link up all of the Supervisor segments as part of the Supervisor bootstrapping operation.



### 4.3 Inter-Task Communication

Inter-task communication refers to the process whereby a given task either stores into or transfers control to another task. Since any such operation is full of dangerous potentialities for violation of task integrity, the Illiac III system imposes a stringently observed strategy upon inter-task communication.

First inter-task communication is decomposed by the Operating System into a sequence of pairwise conversations. In each inter-task link we identify one master (called here the "supervisor") and one slave (called simply the "task"). One supervisor in general will coordinate the activities of several tasks. Typical examples of this sort arise in I/O Interrupts, which send back information for a job in the wait state, and the completion of a subordinate task, which requires sending data to a superior task.

The purpose of this section is first to explain the method by which a task can call upon the supervisor, and second, to describe what mechanisms are available to the supervisor for manipulating the data within the inter-communicating tasks. Finally the process of task activation (and reactivation) is discussed.



#### 4.3.1 Supervisor Calls

SUPERVISOR CALL is an instruction which is executed by a task to transfer control to the supervisor. It is critically important in this operation to protect the supervisor from unauthorized usage whether intentional or accidental. To solve this problem it is necessary to restrict access to the supervisor to a very narrow format. In effect, there is only one entry point to the supervisor.

Three basic steps are involved here:

- 1) The present status of the task (in essence the current contents of its TP fast registers) must be preserved;
- 2) Access to segments (procedure/data) must now be made through the supervisor's segment table rather than the task's segment table; and
- 3) The new status of the supervisor must be loaded into the hardware registers.

The SUPERVISOR CALL instruction does not itself change the contents of the Task Register. The old task name can be retained, or modified, at the discretion of the subsequent supervisor instructions.

Upon completion of the above steps the Supervisor will resume executing instructions -- normally a supervisor procedure called the SVC Handler.

The geometry of transitions to be executed can best be explained by reference to Figure 4.3.1/1. Upon receipt of a SUPERVISOR CALL the basic three steps translate as follows:

- 1) A "snapshot" of the TP fast registers is stored in the task's status segment. A hardwired routine is used to implement this status storage. Note that both the task's status segment and the Supervisor's status segment are local segments in their respective segment tables. Both are assigned the same (wired-in) internal name.

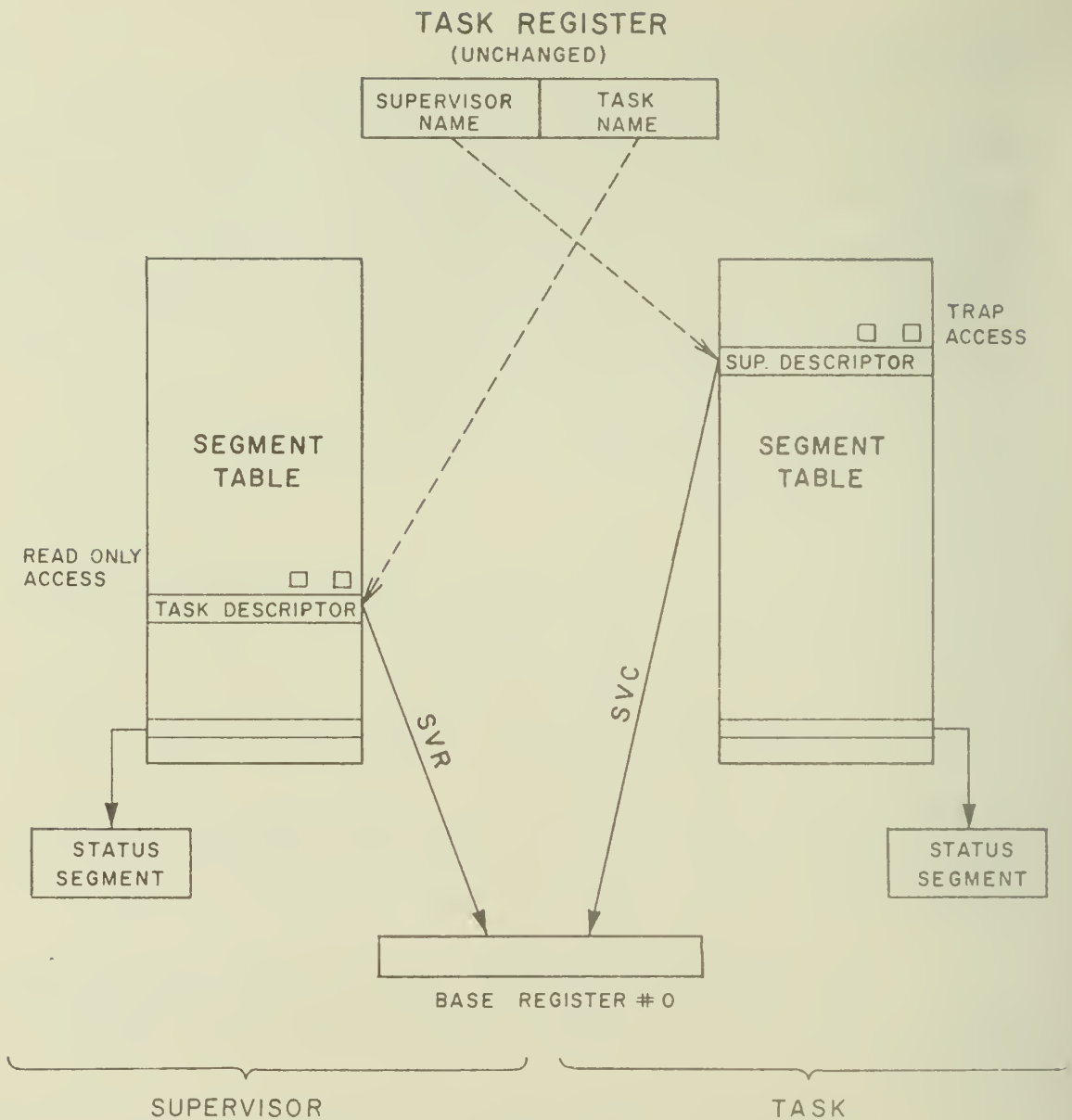


Figure 4.3.1/1 Schematic of a Supervisor Call/Return

- 2) The supervisor name (righthand halfword in the Task Register) is used as an internal segment name to access a descriptor from the Task's segment table. This descriptor, containing the base and bounds of the supervisor's segment table, is then loaded into BR#0. Then the OK bits to all the associative registers are set to 0. At this point the TP is totally under the influence of the supervisor's segment table and is no longer controlled by the task which initiated the call.
- 3) PR#0 is loaded with the SVC Handler internal segment name and virtual address. Instruction execution is reactivated thus effecting the desired transfer of control to the supervisor.

The preservation of task autocracy requires that none of the supervisor's segment table entries are available to task programs. Thus a task cannot directly access any of the Operating System programs without first going through the proper access procedure (i.e. executing a SVC instruction). This isolation is insured if the descriptor of the supervisor's segment table, as stored in the task's segment table, has its access bits set to "trap". Then a task procedure, attempting to access an entry in the supervisor's segment table, necessarily will always incur an automatic interrupt. Nor can this situation be changed by the task alone. The task program cannot load or modify either its segment table or its base registers: every access the task makes is under control of its table. At the same time it is relatively easy for the machine to get to the supervisor since it can modify its base registers.

The task must also be constrained from constructing a pseudo "Supervisor's segment table", (and associated status segment), and thus transferring control (via an SVO instruction) to this creation of its own imagination. For to permit this would allow the unscrupulous user to successively scan all of memory and wreck appropriate havoc. For this reason the supervisor's segment table descriptor is stored at a "fixed"



address, i.e. at an address not subject to modification by the task itself. The supervisor name field of the task register contains this internal segment name. Accordingly instructions which would modify the task register are privileged, i.e. not executable by a task.

On the other hand it is quite essential that the supervisor be able to add or modify entries in the task's segment table/status segment, or on occasion construct these ab initio. Examples here include assigning new segment locations (paging), augmenting the bounds of segments previously available, processing assorted interrupts or, more generally, constructing these task tables prior to task activation. For those reasons the descriptor of the task's segment table (i.e. the old contents of BR#0), as stored in the supervisor's segment table, must permit unrestricted (read/write) access.

Upon execution of the SUPERVISOR CALL the first operation of the supervisor is to process the call for legality. If it is acceptable, the supervisor proceeds with the requested function. The SVC Handler uses as parameters the contents of the various PR's which were stored in the task's status segment. This technique allows the system programmers ample flexibility and at the same time gives the task program a minimum degree of control over the SVC instruction itself.

An immediate concomitant of the above technique however is that the task's status segment be a segment, or more elegantly, embedded in a segment available to the supervisor. Most conveniently all status segments of subservient tasks can be packaged as blocks of one supervisor segment: the task status table.

As long as the supervisor processing is concerned with the task which made the original SVC, the Task Register will contain the name of that task. Thus if the task requests an I/O operation, the name of the requesting task will be readily available. During this time operations performed are considered to be part of (and can be charged to) the requesting task.

At a certain point, however, it may become necessary for the supervisor to halt operation on the requesting task. This situation might arise, for example, if the task must await the completion of an I/O command. In this case, the job will be put in a wait state and the processor will either be appropriated to service some other job (in which case its Task Name will be loaded into the Task Register) or be in-activated.



#### 4.3.2 Supervisor Manipulation of Task Segments

Once the supervisor task has gained control, it will, of course, be necessary for it to manipulate data. However, the supervisor can only directly access those segments which have descriptors in the supervisor's Segment Table. The Supervisor, however, must be able to manipulate data contained in user segments. The total number of these user segments in the system could quite possibly exceed the number of possible entries in the supervisor task's Segment Table. Accordingly, some means must be found to allow the supervisor task to access those particular segments with which it is concerned at any given instant. The means for doing this is provided by the RENAME instruction.

The basic idea behind the renaming process is to allow the supervisor to load the descriptor of a user task's segment into a free position in the dynamic portion of the supervisor's Segment Table (see Figure 5.1.2/1). The supervisor can then manipulate the segment in any way desired by using its own internal segment name for the segment. When it has finished using the segment, the supervisor can return the temporary descriptor cell to a free list, using the technique described in Section 5.1.2.

The RENAME instruction utilizes two pointer registers and the OS. One PR serves as the available space pointer register. The other points to an address containing a base descriptor. The double word descriptor is loaded into the cell obtained from the available space segment and the virtual address of the segment is then loaded into the OS. From there the address, which is actually the new internal segment name assigned to the segment by the supervisor, can be used for whatever purposes the supervisor requires.

Whenever the supervisor determines that it is through using a given segment, it can free the descriptor entry in the Dynamic Segment List simply by executing a PUT instruction with the proper available space pointer.



### 4.3.3 Supervisor Return

Eventually the Supervisor will complete the requested operation and will need to return control to the task which originally made the request. This is done with the SUPERVISOR RETURN instruction. As a result of this operation, BR#0 and all of the PR's will be reloaded with the original information which was present when the task made its SVC. At the same time the protected instruction flip-flop is set to zero so that the activated task will not be able to perform protected instructions and the OK bits are set to zero.

The Supervisor Return (SVR) instruction mirrors in its execution that of the SVC--with this essential difference: a "snapshot" of the TP hardware registers at the time of the SVR encounter is not saved in the supervisor's status segment. On the other hand the BR#0 descriptor swapping and the reloading from the task's status segment into the TP hardware registers follows the now familiar pattern. Note however that the new BR#0 descriptor is retrieved in this case from the supervisor's segment table using the task name.

SUPERVISOR RETURN can also be used by the Supervisor to initiate subservient tasks. In this case the supervisor first builds an image of the desired task's segment table and appropriate startup configuration of the hardware registers. Execution of the SVR transfers control to the task. Again in this situation status of the supervisor hardware at the time of the SVR is irrelevant and not saved.





#### 4.4 System Handling of Interrupts

The Illiac III Supervisor system must be able to respond to two types of interrupt conditions: local interrupts and distal interrupts. The local interrupts concern conditions originating within the TP or as a direct consequence of a TP command to a unit: AU, PAU or Core. These interrupts are also called traps. Examples include loss of significance in a floating point AU operation, an illegal plane address in a PAU instruction, or a bounds overflow in addressing core.

Distal interrupts are caused by some external processor and are routed to a TP through the interrupt unit. Examples of this type of inter-processor communication include the ACTIVE TP instruction, I/O interrupts, etc.

Interrupts are processed in several stages. Upon detection of an interrupt, it is the responsibility of the Taxicrnic Processor to save the interrupt information and the current hardware status of the machine in the Interrupt Storage Segment, and to transfer control to the current task's Interrupt Handler Procedure. This first step is entirely hardware implemented.

The Interrupt Handler Procedure, in turn, determines the type of interrupt which has occurred on the basis of the information stored in the Interrupt Storage Segment and transfers control to the appropriate procedure which will process the interrupt. NOTE: This latter processing procedure may be either a routine belonging to the present task or it may be a Supervisor routine. Interrupts do not necessarily imply Supervisor intervention.

If a Supervisor routine must be called to handle the interrupt an SVC instruction is used. If the Supervisor subsequently decides that the interrupted task must be taken off of the machine (either temporarily or permanently) it fills the task's Status Segment with the necessary task information to enable the task to be reactivated later on (if possible), and then deactivates the task.

The Illiac III interrupts themselves have been divided into several priority levels. The interrupts are recognized and processed on the basis of these priorities, i.e. if the TP is processing a given level of interrupt, all lower level interrupts are masked off. The masking is controllable by both hardware and software.

The purpose of this section is to explain in some detail what will happen at each processing stage in the handling of interrupts. Although much of the information found in this section is repeated elsewhere, it was felt that a proper understanding of the Operating System requires a unified description of interrupts.

A list of local interrupts which might be detected by the Taxicrinc Processors is given in Figure 4.4.

TP            Interrupts:

- BOV    -   bounds overflow
- ILAC   -   illegal access mode
- SUF    -   stack underflow
- ILI    -   illegal instruction
- AOV    -   adder overflow
- ASE    -   available space empty
- ILEX   -   illegal EXIT
- PVV    -   privilege violation

AU            Interrupts:

- OV     -   overflow
- UN     -   underflow (FLT)
- LS     -   loss of significance (FLT)
- ID     -   invalid decimal (BCD)

PAU           Interrupts:

- PAUI   -   PAU interrupt

IU            Interrupts:

EN            Interrupts:

- ILAD   -   illegal address
- PAR    -   parity error

Figure 4.4    Local Interrupts Detected by the TP's



#### 4.4.1 TP Operation Upon Interrupt Detection

As mentioned in the previous section, a TP can respond to two types of interrupts, local and distal. Local interrupts are generated within the TP and are generally handled by the task which generated the interrupt. Distal interrupts are caused by some processor or peripheral device and are generally routed to the TP through the Interrupt Unit. The purpose of this section is to describe the operation of the TP when it is confronted with one of these two types of interrupts.

The Taxicrinic Processor has one major complication with regard to interrupts which is not found in most CPU's, namely that many instructions perform irrevocable changes on the process data base before the existence of an interrupt (or trap) is determined. Thus if an interrupt does occur, it is not possible to merely stop the execution of the instruction and after the interrupt has been satisfied to begin at the beginning. Instead the state of the machine at some mid-point in the instruction must be saved so that the instruction can continue where it left off.

In the general case, this situation would require hundreds of bits simply to designate the sequence path which the instruction had taken at the time of the interrupt. Luckily, however, even though the instruction as a whole may not be restartable, there are many sequences which can be restarted if a local interrupt occurs within them. At the same time, since the TP has some flexibility in deciding when to recognize distal interrupts, it can always avoid looking for them during sequences which are "restartable". As a result, the number of possible "restarting points" can be greatly reduced since the TP will never have to return from an interrupt to any of the middle points of a restartable sequence.

Thus, in the design of the TP sequences themselves careful attention is paid to the sequences in which it is possible for a local interrupt to occur. For every such situation every effort is made to

either 1) classify the interrupt as catastrophic (i.e. there will be no return to complete the instruction) or 2) write the sequence in such a manner that the interrupt will be detected before any permanent changes are made to the memory or the pointer or base registers by that sequence. If this can be done then whenever a local interrupt is detected, a special "interrupt exit" can be made back to the calling sequence. This means that the sequence which detected the interrupt will be restarted when operation is resumed after the interrupt handling. By going through this process at higher and higher levels of nested calls, the interrupt can be bucked back until eventually a return is made to a sequence which cannot undo a previous operation. Such a point in a sequence is called an Interrupt Point. After the interrupt has been processed, control will have to be returned to the proper Interrupt Point in order for the interrupted instruction to continue. In addition, these Interrupt Points are the only points in the control sequences where the TP looks for distal interrupts.

Once an interrupt has been detected and control has reached an Interrupt Point, the hardware transfers control to the hardware Interrupt Control Sequence. This sequence has the responsibility for determining the level of the recognized interrupt and for storing the information necessary for processing the interrupt in the Interrupt Storage Segment. In addition it must save the necessary TP registers and status information which will be needed to restart the TP when processing resumes after the interrupt has been taken care of. The actual storage process consists of accessing the **last filled block control double word** of the appropriate interrupt level to obtain a pointer to the storage block which is to be used (See Section 4.4.2). Then the TP makes write accesses to successive locations within the block until all the necessary information has been stored.

The information stored by the TP consists of two types: information needed to process the interrupt, and information used to restore the status of the TP when the interrupt processing has been completed. In order to keep the volume of the latter type of information as small as possible, only those registers are saved which are absolutely



needed to specify the interrupt. If more registers are needed later on in the interrupt processing, the procedure performing the processing has the responsibility for saving their contents. As a result among the Pointer Registers, for example, only PR#0 and PR#1 are initially saved in the storage block. PR#0 must be used in transferring control to the Interrupt Handler, while PR#1 is used to point to the storage block itself. (For the storage block format, see Figure 4.4.1).

In addition to holding the two types of information just mentioned, the Storage Block is also used as temporary storage by the Interrupt Handler. This is necessary since the Interrupt Handler Procedure cannot necessarily assume that any data storage other than the Storage Block itself is still usable (i.e. Available Space may be exhausted, the Operand Stack may have had a bounds overflow, etc.). The TP will load one predetermined part of this area of the Storage Block with the entry which was used in the pointer circular buffer. This allows the Interrupt Handler to replace the original pointer entry with a new pointer to a new unused block of storage.

Once these storage operations have been completed, the TP has finished the operations necessary for the recognized interrupt.

It is possible, of course, that more than one interrupt was present at the time the TP recognized the interrupt situation. In such cases the TP chooses the highest priority interrupt, or if there is more than one at the highest priority it will choose one of them by some deterministic method. All other interrupts will remain pending.

After having stored the necessary information, the TP turns on the interrupt masks and transfers control to the appropriate processing level of the Interrupt Handling Procedure. The transfer of control is accomplished by loading PR#0 with the standardized segment name for the Interrupt Handler Procedure and the virtual address of the appropriate entry in the Interrupt Handler's branch table. This location will contain a branch instruction to the procedure which takes care of interrupts at the given level.



0	BLOCK POINTER	FOR USE BY SYSTEM		
8	SEGMENT NAME	VIRTUAL ADD.	LR	
16	DR		PR# 0	
24	PR# 1		PRSNR# 0	PRSNR# 1
32	SBR		AR	
40	TGR		INST. MNEM.	CONTROL FF STATUS
48	CALLING CONTROL POINT STATUS			
56	PRIORITY LEVEL	INTERRUPT TYPE		IR

Figure 4.4.1 Interrupt Storage Block Format

#### 4.4.2 Interrupt Storage Segment

Upon detection of an interrupt, it is the responsibility of a Taxicrinic Processor to store certain registers, status flip-flops, and other indications of its current state along with enough information to tell the Interrupt Handler Procedure what type of interrupt took place. The purpose of this section is to describe the Interrupt Storage Segment (which contains space for this information), and to explain how information is loaded into and out of this storage area.

The Interrupt Storage Segment is unique for each task in the Illiac III system. Although the length of this segment can vary from one task to the next, depending on the expected amount and type of interrupt activity, the basic format is the same. As shown in Figure 4.4.2/1, the Interrupt Storage Segment is divided into three basic areas: the status area, the pointer circular buffer area and the storage block area.

The status area is divided into entries corresponding respectively to the various interrupt levels. Each entry consists of a Last Filled Block control double word (CDW) which indicates the status of the pointer circular buffer for the corresponding interrupt level. It indicates the pointer in the circular buffer which points to the last block to be filled with interrupt information at that interrupt level. At the end of the status area there is, in addition, an available space format entry which is used by the Interrupt Handler Procedure in assigning storage blocks to the various circular buffers.

The pointer circular buffer area is also divided into sections corresponding to the interrupt levels. Each section contains one circular buffer of pointer entries. These buffers may contain anywhere from one entry on up.

The storage block area comprises the rest of the Interrupt Storage Segment and consists of storage blocks used to contain

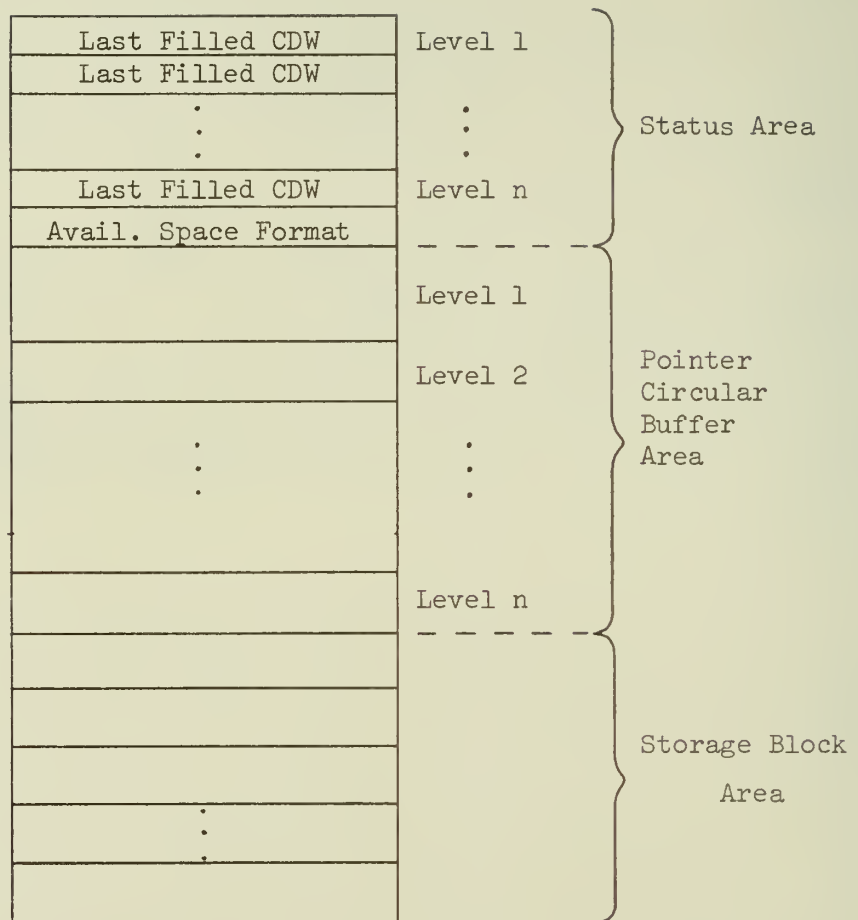


Figure 4.4.2/1 Interrupt Storage Segment Structure

the information to be stored during interrupts. These blocks, which are controlled by the standard available space techniques, are shifted on to queues, processed, and eventually returned to available space.

As far as using the Interrupt Storage Segment is concerned, all storing of interrupt information is controlled through the control double words. The CDW's have the format shown in Figure 4.4.2/2. The rightmost halfword contains the pointer to either the last filled or last emptied entry in the corresponding level's circular buffer. The leftmost halfword field contains the beginning address of the last entry before the beginning of the circular buffer. The second halfword contains the increment field. The third halfword contains the beginning address of the last entry in the corresponding level's circular buffer.

Given this format, the process of finding storage for the interrupt information at a given level interrupt is quite simple (refer to Figure 4.4.2/3). First an access is made to the Last Filled Block control double word. The pointer thus obtained is then incremented by the length of an entry in the circular buffer (the increment field of the CDW). The new pointer value is used as the address of the pointer to the block into which the interrupt information is to be stored. If the new pointer value does not equal the maximum allowed address, it is stored in the pointer field of the control doubleword. Otherwise it is still used to find the block for storing the current interrupt information but, instead of the incremented value, the initial value from the leftmost halfword field is stored in the pointer location in the control word.

An important point to note is the case of the Supervisor which can in fact be running on several TP's at one time. In this case, or in general in the case of any task which may run on more than one TP at a time, a problem of processor interference during interrupts occurs. If two "Supervisor" TP's (i.e. TP's on which the Supervisor is running) both have interrupts at the same level at the same time,

Initial Value	Increment	Maximum Value	Pointer Value
------------------	-----------	------------------	------------------

Figure 4.4.2/2 Control Double Word Format

CDW:

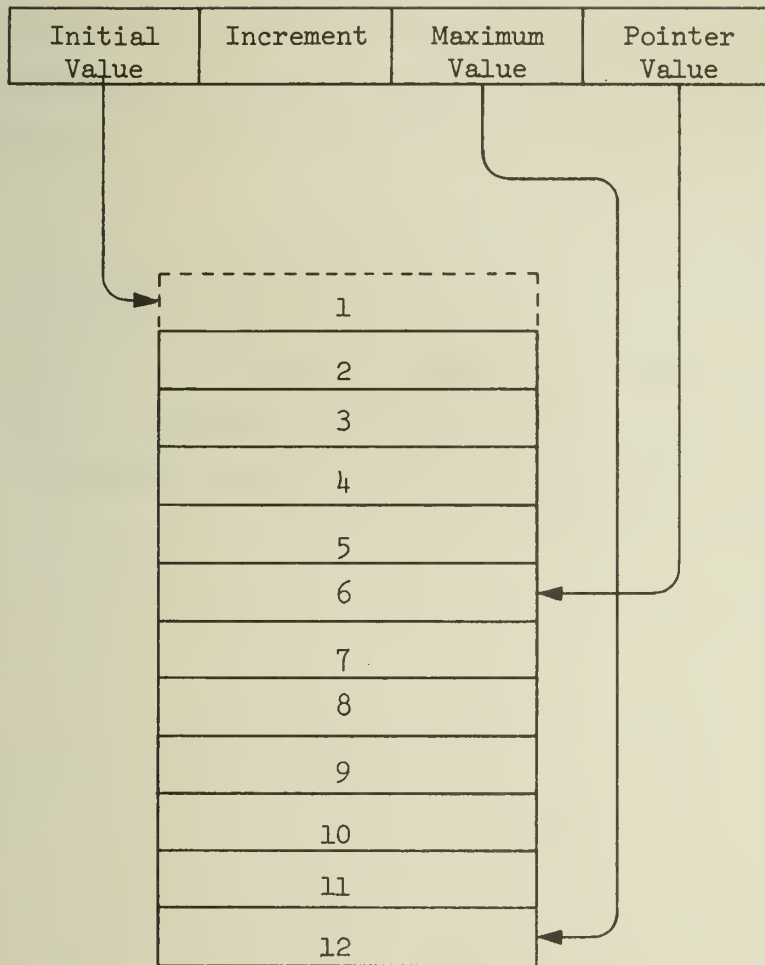


Figure 4.4.2/3 Use of CDW to Obtain an Interrupt Storage Block

they could end up both filling the same storage block in the Interrupt Storage Segment. To avoid this situation it is necessary for the control word manipulations to be performed during only one Exchange Net access. Since the Exchange Net does not allow more than one TP to access a given core box at one time, this restriction will ensure that by the time the second TP gets to the Last Filled Block control double word, it will have been updated to reflect the use of one of the storage blocks by the first TP. The only operation necessary on the part of the first TP is that it must not let go of the Exchange Net until after it has updated the pointer entry and replaced it. This is done using the logic of the Increment and Check (INCK) instruction.



#### 4.4.3 Interrupt Handler Procedure

The system Interrupt Handler is a global procedure which must be included in every task in the machine. It is given a standard internal segment name, namely the name of the second entry in the Global Segment Table.

The Interrupt Handler itself performs any queueing operations needed for the given level interrupt, determines which interrupt in fact took place, and transfers to an appropriate procedure to take care of it. The Interrupt Handler is divided into sections corresponding to the various interrupt levels. At the beginning of the procedure is a jump table consisting of successive jumps to the various processing levels. This allows for easy transfer to the desired section simply by jumping to a relative location in the Interrupt Handler Procedure which is a direct linear function of the interrupt level.

As mentioned in Section 4.4.1, the TP hardware transfers to the Interrupt Handler Procedure's branch table with all interrupts masked. This allows the Interrupt Handler to manipulate any necessary queues without having to worry about being interrupted in the middle of the operation. Of course it is important that this period of time during which all interrupts are masked be as short as possible. If it turns out that no queue operations are necessary for that type of task and that particular interrupt level, then the first instruction in that section of the Interrupt Handler will be one which unmask all interrupts of greater or equal priority.

The first operation involved in queueing is performed after control transfers from the branch table to the appropriate interrupt section. The task here is to set up the pointer circular buffer so it can accept a new interrupt. If it is possible for another interrupt to occur at the same level before the current one has been processed, it will be necessary to replace the appropriate entry in the appropriate circular buffer with a pointer to a new storage block. This is done using

a GET instruction and the available space format entry at the end of the Status Area of the Interrupt Storage Segment. The new block pointer is loaded into the entry whose address has been stored by the TP in the current storage block. It may be necessary (see below) to use a INCK instruction while the available space entry is being modified, if more than one processor is working on the task.

Once the circular buffer has been restored, if necessary, the operations for queueing the interrupt have been completed. PR#1 points to the interrupt block and thus any further data accesses can be executed through it. If another interrupt should occur later, PR#0 and PR#1 will be saved in a new interrupt block and when that interrupt is eventually finished, and an interrupt return made, processing can then continue on the earlier one.

As soon as the Interrupt Handler unmaskes the appropriate interrupt levels there may be another interrupt. This would occur, for example, if the TP had detected more than one interrupt at the same level to begin with or if during the process of handling the interrupt a new external interrupt had occurred<sup>1</sup>. At any rate, if an interrupt of equal or higher priority is pending at the time the interrupt masks are set to zero, a new interrupt will occur and the process will begin anew. If further interrupts do occur, these successive interrupts will be processed in the reverse order of their order of recognition. This is all right, however, since these interrupts must be of equal or greater priority than the first anyway.

From the description of this procedure, it will be noted that as long as only one TP is performing processing on a given task (either on the task itself or on interrupts detected by the task), there will never be a need to have a circular buffer with more than one pointer in it. This is because the Interrupt Handler will always reload the buffer with a pointer to an unused storage block if there is any chance

---

<sup>1</sup> Hopefully the Interrupt Handler will not produce an interrupt of its own, at least not during this initial queueing operation.

that there will be another interrupt at the same level before the first interrupt has been completely processed. The use of pointers in the buffer allows this unloading to be much faster than if the data itself were stored in the buffer. Circular buffers with more than one pointer entry are only needed in those tasks, e.g. the Supervisor, which may be running on more than one processor at a time. In such cases there needs to be one entry for each possible TP which may be processing the task simultaneously.

In some cases the processing involved in handling an interrupt will involve a transfer of control to the Supervisor by means of an SVC instruction. In others it will mean transferring to some other intra-task segment. In this latter case the transfer of control (since it is going to an external segment) is under control of the Interrupt Handler Procedure's own linkage table. This gives the Operating System a means of controlling the transfer.

We recall that in a normal inter-segment linking process (see Section 4.2) the linkage table remains empty when it is first formed and is only filled as needed. Now suppose that at the time the task is created, its Interrupt Handler linkage table, instead of being left empty, is initially loaded with entries linked to the specific procedures that task will use in case of an interrupt. These segments must be known at task initiation time since the compiler or assembler which wrote the instructions has to specify any unusual interrupt conditions. In this manner any nonstandard routines can be automatically inserted if they are requested.

As a result of the preloading of the interrupt procedure's linkage table there need never be a linkage fault on the interrupt handling segment and this is in fact the desired result. If later on in the execution of the task, the task might wish to change one of its interrupt handling procedures, this change would be perfectly acceptable and could be done by having the supervisor modify the proper entry in the linkage table of the task's interrupt handler.

A special situation arises in the case of interrupts which occur while the Supervisor Task is in control of the TP. In particular, if the interrupt necessitates making a Supervisor Call, then a problem might develop when the interrupt operation is over and a Supervisor Return is executed. In this case the Protected Instruction flip-flop will be reset to zero. But since the TP is returning to another part of the Supervisor Task, problems will develop as soon as the TP executes a protected instruction.

Two possible solutions to this problem were considered:

- 1) Design the SVC instruction so that it could be recursive and stack the status (supervisor/non-supervisor) of the calling task.
- 2) Rewrite the Interrupt Handler for the Supervisor Task so that an interrupt will never produce an SVC instruction. If the need for Supervisor handling of an interrupt within the supervisor occurs, a normal CALL instruction will be executed. This bypasses the SVC Handler and directly calls the needed supervisory procedure.

The first solution was judged to be too expensive to implement in hardware and would needlessly complicate the SVC instruction for a (relatively) few number of cases. The main problem with the second solution is that the Supervisor Task will have a different Interrupt Handler than the other tasks, not just a different Interrupt Handler linkage table. This is no great problem though, since it can still use the standard Interrupt Handler entry in the Supervisor Task Segment Table.

#### 4.4.4 Interrupt Return Operation

Once the Interrupt Handler Procedure has determined the type of interrupt which has occurred and has completed a call to the appropriate processing procedures, it executes an Interrupt Return instruction. This instruction uses PR#1 as an operand since it will still point to the Interrupt Storage Block containing the return status information. The execution of this instruction will cause the TP hardware to restore its status before the interrupt. Just prior to resuming operation, the TP will also restore the interrupt storage block to the available space list.

Several points should be mentioned. First of all the Interrupt Mask will be maintained by having it as one of the items saved in the interrupt block. In this manner the state of the mask when the TP begins processing at the return location will be automatically the same as before.

Secondly, the problem of security must be considered. There is always a certain amount of uncertainty as to whether or not the operand in the Interrupt Return instruction points to a valid Interrupt Return block. If it does not, or if it is pointing to the wrong block, all hell may break loose when the TP loads up its "new status". The main problem seems to be how to keep the area of core used for Interrupt Block storage from being written on by anyone else except the TP hardware and the Interrupt Handler Procedure (which had better be "fully debugged").





U.S. ATOMIC ENERGY COMMISSION  
UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR  
DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

( See Instructions on Reverse Side )

AEC REPORT NO. 472 COO-2118- 0019	2. TITLE ILLIAC III REFERENCE MANUAL VOLUME IV: Supervisor Organization
--------------------------------------	---

TYPE OF DOCUMENT (Check one):

☒

a. Scientific and technical report

☐ b. Conference paper not to be published in a journal:

Title of conference \_\_\_\_\_

Date of conference \_\_\_\_\_

Exact location of conference \_\_\_\_\_

Sponsoring organization \_\_\_\_\_

☐ c. Other (Specify) \_\_\_\_\_

RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

☒

a. AEC's normal announcement and distribution procedures may be followed.

☐

b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.

☐

c. Make no announcement or distribution.

REASON FOR RECOMMENDED RESTRICTIONS:

SUBMITTED BY: NAME AND POSITION (Please print or type)

Prof. Bruce H. McCormick  
Principal Investigator  
Illiac III Project

Organization

Department of Computer Science  
University of Illinois  
Urbana, Illinois 61801

Signature

Bruce H. McCormick

Date

Aug. 12, 1971

FOR AEC USE ONLY

AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION  
RECOMMENDATION:

PATENT CLEARANCE:

☐

a. AEC patent clearance has been granted by responsible AEC patent group.

☐

b. Report has been sent to responsible AEC patent group for clearance.

☐

c. Patent clearance not required.





SEP 28 1971















UNIVERSITY OF ILLINOIS-URBANA

510.84 IL6R no. C002 no.469-474(1971)

Internal report /



3 0112 088399966